Review: Operator Overloading

- Operators +, - operate on integers
- Write procedures for bit vector addition/subtraction
- addvec, subvec
- Operator overloading allows using + operator to implicitly call an appropriate addition function
- How does it work?
  - When compiler encounters a function declaration in which the function name is an operator enclosed in double quotes, the compiler treats the function as an operator overloading ("+")
  - When a "+" operator is encountered, the compiler automatically checks the types of operands and calls appropriate functions

Review: Multivalued Logic

- Bit (0, 1)
- Tristate buffers and buses => high impedance state 'Z'
- Unknown state 'X'
  - e.g., a gate is driven by 'Z', output is unknown
  - a signal is simultaneously driven by '0' and '1'

Additional Topics in VHDL

- Attributes
- Transport and Inertial Delays
- Operator Overloading
- Multivalued Logic and Signal Resolution
- IEEE 1164 Standard Logic
- Generics
- Generate Statements
- Synthesis of VHDL Code
- Synthesis Examples
- Files and Text IO
Review: Signal Resolution

- VHDL signals may either be resolved or unresolved
- Resolved signals have an associated resolution function
- Bit type is unresolved –
  - there is no resolution function
  - if you drive a bit signal to two different values in two concurrent statements, the compiler will generate an error

Review: Signal Resolution (cont’d)

signal R : X01Z := 'Z'; ...
R <= transport '1' after 2 ns, 'Z' after 6 ns;
R <= transport '1' after 8 ns, '0' after 10 ns;

IEEE 1164 Standard Logic

- 9-valued logic system
  - 'U' – Uninitialized
  - 'X' – Forcing Unknown
  - '0' – Forcing 0
  - '1' – Forcing 1
  - 'Z' – High impedance
  - 'W' – Weak unknown
  - 'L' – Weak 0
  - 'H' – Weak 1
  - '-' – Don’t care
  - If forcing and weak signal are tied together, the forcing signal dominates.
  - Useful in modeling the internal operation of certain types of ICs.
  - In this course we use a subset of the IEEE values: X10Z

Review: Resolution Function for X01Z

package Reso(X): type X01Z is (0, 1, 'Z'); end Reso;
define AND and OR for 4-valued inputs?

package both(Reso): type X01Z_01 is (0, 1); end both;
Resolution Function for IEEE 9-valued

```vhdl
constant resolution_table : std_logic_table := (
  -- | U | X | 0 | 1 | Z | 1W | 1L | 1H |
  -- | 0'1 | 1'0 | 0 | 0'1 | 0 | 0 | 0'1 | 0'1 |
  -- | 1'0 | 1 | 0'1 | 0 | 0 | 0 | 0 | 1'0 |
  -- | 0'1 | 0 | 0 | 0'1 | 0 | 0 | 0 | 1'0 |
  -- | 1W | 0 | 0 | 0 | 0 | 0 | 0 | 1'0 |
  -- | 1L | 0 | 0 | 0 | 0 | 0 | 0 | 1'0 |
  -- | 1H | 0 | 0 | 0 | 0 | 0 | 0 | 1'0 |
);```

AND Table for IEEE 9-valued

```vhdl
constant and_table : std_logic_table := (
  -- | U | X | 0 | 1 | Z | 1W | 1L | 1H |
  -- | '1'0 | 1'0 | 0 | 0 | 0 | 0 | 0 | 1'0 |
  -- | 1'0 | '1'0 | 0 | 0 | 0 | 0 | 0 | 1'0 |
  -- | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1'0 |
  -- | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1'0 |
  -- | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1'0 |
  -- | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1'0 |
  -- | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1'0 |
);```

AND Function for std_logic_vectors

```vhdl
function "and" ( l : std_logic_vector ; r : std_logic_vector ) return std_logic is
begin
  return (and_table ( l'length ));
end "and";
```

Generics

- Used to specify parameters for a component in such a way that the parameter values must be specified when the component is instantiated
- Example: rise/fall time modeling
Rise/Fall Time Modeling Using Generics

entity adder4 is
  port (A, B: in bit_vector(3 downto 0); Cin: in bit; -- Inputs
        S, Cout, Sum: out bit); -- Outputs
end adder4;
architecture Structure of adder4 is
component fulladder
  port (A, B, Cin: in bit; -- Inputs
        S, Cout, Sum: out bit); -- Outputs
end component;
signal C: bit_vector(3 downto 1); -- instantiate four copies of the fulladder
  FA0: fulladder port map (A(0), B(0), C(1), S(0));
  FA1: fulladder port map (A(1), B(1), C(1), S(1));
  FA2: fulladder port map (A(2), B(2), C(2), S(2));
  FA3: fulladder port map (A(3), B(3), C(3), S(3));
end structure;

Generate Statements

- Provides an easy way of instantiating components when we have an iterative array of identical components

Example: 4-bit RCA

<table>
<thead>
<tr>
<th>Cn</th>
<th>Full Adder</th>
<th>Full Adder</th>
<th>Full Adder</th>
<th>Full Adder</th>
<th>Cn</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A(0)</td>
<td>B(0)</td>
<td>Cin</td>
<td>S(0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A(1)</td>
<td>B(1)</td>
<td></td>
<td>S(1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A(2)</td>
<td>B(2)</td>
<td></td>
<td>S(2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A(3)</td>
<td>B(3)</td>
<td></td>
<td>S(3)</td>
<td></td>
</tr>
</tbody>
</table>

4-bit Adder

entity adder is
  port (A, B: in bit_vector(3 downto 0); Cin: in bit; -- Inputs
        S, Cout, Sum: out bit); -- Outputs
end adder;
architecture Structure of adder is
component fulladder
  port (A, B, Cin: in bit; -- Inputs
        S, Cout, Sum: out bit); -- Outputs
end component;
signal C: bit_vector(4 downto 0); -- generate four copies of the fulladder
  FA0: fulladder port map (A(0), B(0), Cin, S(0));
  FA1: fulladder port map (A(1), B(1), Cin, S(1));
  FA2: fulladder port map (A(2), B(2), Cin, S(2));
  FA3: fulladder port map (A(3), B(3), Cin, S(3));
end structure;
Synthesis of VHDL Code

• Synthesizer
  – take a VHDL code as an input
  – synthesize the logic: output may be a logic schematic with an associated wirelist
• Synthesizers accept a subset of VHDL as input
• Efficient implementation?
• Context
  
  \[
  A \leftarrow B \text{ and } C; \quad \text{wait until clk'event and clk = '1'}; \\
  A \leftarrow B \text{ and } C; \\
  \text{Implies CM for } A \quad \text{Implies a register or flip-flop}
  \]

Synthesis of VHDL Code (cont’d)

• When use integers specify the range
  – if not specified, the synthesizer may infer 32-bit register
• When integer range is specified, most synthesizers will implement integer addition and subtraction using binary adders with appropriate number of bits
• General rule: when a signal is assigned a value, it will hold that value until it is assigned a new value

Unintentional Latch Creation

entity tech_example is
  port (a, b : in integer; range 0 to 3;
        c : out boolean);
end entity;

architecture test of tech_example is
begin
  process
    variable G : boolean := 0;
  begin
    case a is
      when 0 => b <= '1';
      when 1 => b <= '0';
      when 2 => b <= '1';
      when others => b <= null;
    end case;
  end process;
end architecture;

What if \( a = 3 \)?

The previous value of \( b \) should be held in the latch, so \( G \) should be 0 when \( a = 3 \).

To eliminate latch // replace the word null with \( b \leftarrow 0 \);

If Statements

if A = '1' then NextState <= 3;
else
  NextState <= 2;
end if;

What if \( A \neq 1 \)?

Retain the previous value for NextState?

Synthesizer might interpret this to mean that NextState is unknown!

if A = '1' then NextState <= 3;
else
  NextState <= 2;
end if;
Synthesis of a Case Statement

```vhdl
entity case_example is
  port(a: in std_logic; b: in std_logic; c: out std_logic)
end case_example;
architecture test of case_example is
begin
  when b => c <= a;
  when b => c <= b;
  when others => c <= c;
end test;
```
• Overloaded operators
  – Unary: abs, -
  – Arithmetic: +, -, *, /, rem, mod
  – Relational: >, <, >=, <=, ==, !=
  – Logical: not, and, or, nand, nor, xor, xnor
  – Shifting: shift_left, shift_right, rotate_left, rotate_right, sll, srl, rol, ror

if the left and right operand sizes are different lengths, the shortest operand will be sign-extended before performing an arithmetic operation. For unsigned operands, the shortest operand will be extended by filling in 0's on the left. Examples:

signed #011011' : '011011' becomes '011011' : '011011' = '011011'
signed #011011' + '011011' becomes '011011' + '011011' = '011011'

When addition is performed on unsigned or signed operands, the final carry is discarded and overflow is ignored. If a carry is needed, an extra bit can be added to one of the operands. Examples:

code: 011011 + 011011  becomes 011011 + 011011  = 011011

In the above example, the notation unsigned("011011") is a type qualification which assigns the type unsigned to the bit vector "011011".

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity example is
  port (signal clk: in bit;
        signal a: in unsigned(3 downto 0): = "0110";
        signal b: in unsigned(3 downto 0): = "0111";
        signal pr: out boolean;
        signal e: out signed(3 downto 0): = "0000"
        signal o: out signed(3 downto 0): = "010001");
end example;

architecture s of example is
begin
  pr <= (a > b); -- 4-bit comparator
  process
  begin
    while (not clk'event and clk = '1') loop
      case (a) is
        when '0' => -- 4-bit regular and 4-bit adder
          e <= e + b;
          o <= o + 1;
      when others => null;
      end case;
    end loop;
  end process;
end;
Synthesis Examples (2a)

- Mealy machine: BCD to BCD+3 Converter

```vhdl
entity SM_1 is
  port(CLK, D: in bit; Z: out bit);
end SM_1;
architecture Table of SM_1 is
signal State, NextState: std_logic_vector(1 downto 0);
constant X1: bit := '0';
constant X2: bit := '0';
constant X3: bit := '0';
constant X4: bit := '0';
constant total: integer := 2;
begin
  process(State, X)
  -- Combinational network
  begin
    State <= NextState <= X0;  -- added to avoid stuck
  case State is
    when X3 =>
      if X = '0' then Z <= '0'; NextState <= '1';
      when X = '1' then NextState <= '0';
    end case;
  end process;
end Table;
```

Synthesis Examples (2b)

- Mealy machine: BCD to BCD+3 Converter

```vhdl
when X3 =>
  if X = '0' then Z <= '0'; NextState <= '1';
  when X = '1' then NextState <= '0';
when X4 =>
  if X = '0' then Z <= '0'; NextState <= '1';
  when X = '1' then NextState <= '0';
when X5 =>
  if X = '0' then Z <= '0'; NextState <= '1';
  when X = '1' then NextState <= '0';
when others =>
  when others => null;
end case;
```

Files

- File input/output in VHDL
  - Used in test benches
    - Source of test data
    - Storage for test results
  - VHDL provides a standard TEXTIO package
    - read/write lines of text
Files

File Declaration

file file-name: file-type [open mode] is "file-pathname";

Example:

file test_data: text open read_mode is "C:\data\test.dat";

• declares a file named test_data of type text which is opened in the read mode. The
default location of the file is in the test1 directory on the c: drive.

Modes for Opening a File

read_mode file elements can be read using a read procedure
write_mode new empty file is created; elements can be written using a write procedure
append_mode allows writing to an existing file

Standard TEXTIO Package

• Contains declarations and procedures for working with files composed of lines of text
• Defines a file type named text:
  type text is file of string;
• Contains procedures for reading lines of text from a file of type text and for writing lines of text to a file

Reading TEXTIO file

• Reading reads a line of text and places it in a buffer with an associated pointer
• Pointer to the buffer must be of type line, which is declared in the textio package as:
  type line is access string;
• When a variable of type line is declared, it creates a pointer to a string
• Code
  variable buff: line;
  ... read_line (test_data, buff);
  – reads a line of text from test_data and places it in a buffer which is pointed to by buff

Extracting Data from the Line Buffer

• To extract data from the line buffer, call a read procedure one or more times
• For example, if bv4 is a bit_vector of length four, the call
  read (buff, bv4)
  – extracts a 4-bit vector from the buffer, sets bv4 equal to this vector, and adjusts the pointer buff to point to the
  next character in the buffer. Another call to read will then extract the next data object from the line buffer.
Extracting Data from the Line Buffer (cont’d)

- TEXTIO provides overloaded *read* procedures to read data of types bit, bit_vector, boolean, character, integer, real, string, and time from buffer
- Read forms
  - `read(pointer, value)`
  - `read(pointer, value, good)`
    - `good` is boolean that returns TRUE if the read is successful and FALSE if it is not
    - type and size of value determines which of the read procedures is called
    - character, strings, and bit_vectors within files of type text are not delimited by quotes

Writing to TEXTIO files

- Call one or more write procedures to write data to a line buffer and then call writeln to write the line to a file
  - Example:
    ```
    variable buffw : line;
    variable int1 : integer;
    variable bv8 : bit_vector(7 downto 0);
    ...  
    writeln(buffw, output_file);
    ```
  - Write parameters: 1) buffer pointer of type line, 2) a value of any acceptable type, 3) justification (left or right), and 4) field width (number of characters)

An Example

- Procedure to read data from a file and store the data in a memory array
- Format of the data in the file
  - address N comments
  - byte1 byte2 ... byteN comments
  - Address - 4 hex digits
  - `N` indicates the number of bytes of code
  - bytei - 2 hex digits
  - each byte is separated by one space
  - the last byte must be followed by a space
  - anything following the last state will not be read and will be treated as a comment

An Example (cont’d)

- Code sequence: an example
  - `12AC 7` (7 hex bytes follow)
    - AE 03 B6 91 C7 00 DC (LDX imm, LDA dir, STA ext)
    - 005B 2 (2 bytes follow)
    - 01 FC_
  - TEXTIO does not include read procedure for hex numbers
    - we will read each hex value as a string of characters and then convert the string to an integer
  - How to implement conversion?
    - table lookup – constant named lookup is an array of integers indexed by characters in the range ‘0’ to ‘F’
      - corresponding values:
        - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
Things to Remember

• Attributes associated to signals
  – allow checking for setup, hold times, and other timing specifications
• Attributes associated to arrays
  – allow us to write procedures that do not depend on the manner in which arrays are indexed
• Inertial and transport delays
  – allow modeling of different delay types that occur in real systems
• Operator overloading
  – allow us to extend the definition of VHDL operators so that they can be used with different types of operands.

Things to Remember (cont’d)

• Multivalued logic and the associated resolution functions
  – allow us to model tri-state buses, and systems where a signal is driven by more than one source
• Generics
  – allow us to specify parameter values for a component when the component is instantiated
• Generate statements
  – efficient way to describe systems with iterative structure
• TEXTIO
  – convenient way for file input/output
State Graphs for Control Networks

- Use variable names instead of 0s and 1s
  - E.g., XIXj/Z0Z4
    - If Xi and Xj inputs are 1, the outputs Zp and Zq are 1 (all other outputs are 0s)
  - E.g., X = X1X2X3X4, Z = Z1Z2Z3Z4
    - X1X4/Z2Z3 = 1 - 0 / 0 1 1 0

Constraints on Input Labels

- Assume: I = input expression =>
  we traverse the arc when I=1
  1. If i and j are any pair of input labels on arcs exiting state Si, then I(ij) = 0 if i ≠ j.
     Assures that at most one input label can be 1 at any given time
  2. If n arcs exit state S0 and the n arcs have input labels I1, I2, ..., In, respectively, then I1 + I2 + ... + In = 1.
     Assures that at least one input label will be 1 at any given time

1 + 2: Exactly one label will be 1 =>
the next state will be uniquely defined for every input combination
Constraints on Input Labels (cont’d)

(1) $f(x_1 x_2 y_1 y_2) = 0$
(2) $f(x_1 x_2 y_1 y_2) = 0$
(3) $x_1 + x_2 y_2 = 1$  

Networks for Arithmetic Operations

Case Study: Serial Parallel Multiplier

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>1011 (12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier</td>
<td>1101 (11)</td>
</tr>
<tr>
<td>Partial</td>
<td>001011</td>
</tr>
<tr>
<td>Product</td>
<td>10000111</td>
</tr>
</tbody>
</table>

Note: we use unsigned binary numbers

Block Diagram of a Binary Multiplier

Ad – add signal // adder outputs are stored into the ACC
Sh – shift signal // shift all 9 bits to right
Ld – load signal // load multiplier into the 4 lower bits of the ACC
and clear the upper 5 bits

Multiplication Example

<table>
<thead>
<tr>
<th>Initial</th>
<th>0 0 0 0</th>
<th>M (13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-bit adder</td>
<td>0 1 1 1</td>
<td>(12)</td>
</tr>
<tr>
<td>shift</td>
<td>0 0 0 0</td>
<td>M</td>
</tr>
<tr>
<td>4-bit adder</td>
<td>0 1 0 1</td>
<td>(11)</td>
</tr>
<tr>
<td>shift</td>
<td>0 0 0 0</td>
<td>M</td>
</tr>
<tr>
<td>4-bit adder</td>
<td>0 0 0 1</td>
<td>(10)</td>
</tr>
<tr>
<td>shift</td>
<td>0 0 0 0</td>
<td>M</td>
</tr>
<tr>
<td>4-bit adder</td>
<td>0 0 0 1</td>
<td>(9)</td>
</tr>
</tbody>
</table>

dividing line between product and multiplier
State Graph for Binary Multiplier

Behavioral VHDL Model

library IEEE;
use IEEE.std_logic_1164.all;
entity mult4x4 is
port (CK: in bit; M: in bit_vector(3 downto 0); H: in bit_vector(3 downto 0); out: out bit);
end mult4x4;

architecture behavior of mult4x4 is
begin
process
  signal State: integer range 0 to 9;
  signal ACC: bit_vector(6 downto 0): = "000000";
  signal H: bit_vector(3 downto 0);  -- accumulator
  signal M: bit_vector(3 downto 0);  -- M is bit 0 of ACC
begin
  wait until (State = 9);
  case State is
  when 0 => -- initial State
    if (State = 9) then
      ACC <= "000000";
    end if;
  when 1 =>
    if (State = 9) then
      ACC <= "000000";
    end if;
  when 2 =>
    if (State = 9) then
      ACC <= "000000";
    end if;
  when 3 =>
    if (State = 9) then
      ACC <= "000000";
    end if;
  when 4 =>
    if (State = 9) then
      ACC <= "000000";
    end if;
  when 5 =>
    if (State = 9) then
      ACC <= "000000";
    end if;
  when 6 =>
    if (State = 9) then
      ACC <= "000000";
    end if;
  when 7 =>
    if (State = 9) then
      ACC <= "000000";
    end if;
  when 8 =>
    if (State = 9) then
      ACC <= "000000";
    end if;
  when 9 =>
    if (State = 9) then
      ACC <= "000000";
    end if;
  when others =>
    if (State = 9) then
      ACC <= "000000";
    end if;
  end case;
end process;
end behavior;

Behavioral VHDL Model (cont'd)